CS180 Midterm

Spenser Wong

TOTAL POINTS

71 / 100

QUESTION 1

1 Problem 1 14 / 16

✓ - 2 pts **You were almost there, buddy! Some tiny errors / absence of a sound or a complete proof / silly errors etc.**

QUESTION 2

2 Problem 2 6 / 16

✓ - 4 pts **Proof missing/wrong**

✓ - 6 pts **The given algorithm doesn't work for most of the test cases / No proper explanation / Inefficient, but points awarded for creativity**

QUESTION 3

3 Problem 3 16 / 16

✓ - 0 pts **Correct**

QUESTION 4

4 Problem 4 3 / 16

✓ - 13 pts **(a). wrong algorithm**

QUESTION 5

5 Problem 5 12 / 16

✓ - 4 pts **Attempt with essentially right ideas and some form of pseudocode or analysis or data structures.**

QUESTION 6

6 Problem 6 20 / 20

✓ - 0 pts **congratulations! good answer**

Name:

Student ID:

CS180 Winter 2018 - Midterm

Wednesday, February 21, 2018

You will have 110 minutes to take this exam. This exam is closed-book and closed-notes. There are 6 questions for a total of 100 points. **Please write your name and student ID on every page of your solutions. Please use separate pages for each question.**

Question	Points
1	
2	
3	
4	
5	
6	
Total	

Name:

Student ID

1. **[16 points]** We call stable matching *fair* among n men and n woman if all participants get their best matching among all possible stable matchings. Is there a polynomial time algorithm to find if a fair stable matching exists? If so, please describe it, prove it correct and state it's running time.

Yes there is. I propose we use the Gale-Shapely Algorithm. We have alrealy proven in class that the Gale-Shapely algorithm is proposer optimal, i.e. that a member of the proposing set will always be matched with the best possible valid partner. We have proven that this matching is unique as well. I propose that we run the Gale-Shapely Algorithm first with the n men proposing, and then with the n women proposing. If these two matchings are identical, we prove that all participants get their best possible matching. It is impossible for these matchings to match and not be a fair matching, as this contradicts the earlier notion that each partner gets their best valid partner. We call Gale-Shapely twice, which is $O(n^2)$, and we can compare matchings in $O(n)$ time, so the running time is $O(n^2)$.

1 3 5 7 | 2 4 6 9

4 4

Name:

Student ID:

2. [16 points] You are given two sorted lists of size n_1 and n_2. Give an $O(\log n_1 + \log n_2)$ time algorithm for computing the kth smallest element in the union of the two lists. Prove that your algorithm is correct.

~~For this algorithm, I propose that we use a divide and conquer approach, as log in runtime represents a subdivision of work, and this is the only way to meet the time bound. We already have a subdivision into two lists~~

For $k=1$ or $k=n_1+n_2$, we can simply compare the first element of n_1 with the first of n_2, and return that for $k=1$, or the last element of n_1 with the last of n_2 for $k=n_1+n_2$. For $1<k<n_1+n_2$, we use a different approach. I suggest that we insert the elements of n_1 into a heap and the elements of n_2 into a heap. We can get the k^{th} element by removing the top element of the heap k times. We must insert n_1 elements into the heap, which takes $O(\log n_1)$ and the elements of n_2, which takes $O(\log n_2)$. Removing every element takes $O(\log(n_1+n_2))$, thus our algorithm is $O(\log n_1 + \log n_2)$. A heap is arranged so that the minimum element is always at the top. Thus, as long as we have a properly implemented heap, our algorithm will always be correct. We ensure this by correct heapify up/down operations ∎

~~k~~=2

1 ~~3 5 7~~ | 2 4 6 9 13 11

1 3 | 5 7 ~~2 4 6~~ | 9 13 11

element =2

1 ~~3~~ 5 7 | 2 4 6 8

~~10 11 12 13~~ | 1 2 3 4

Name:

Student ID:

3. **[16 points]** You are given a directed acyclic graph G. Give a linear time algorithm that checks if there exists a Hamiltonian Path in G. (Recall that Hamiltonian Path is a simple path that visits every node exactly once).

We can do this by ~~a~~ reduction to a topological ordering. As proved in the textbook, we can create a topological ordering by finding the node in G that has no incoming edges. We then remove this node to be the next node in the topological ordering and remove all edges incident on this node. ~~We continue this process until~~ and place them into the topological ordering. We continue this process until all edges and nodes are in the topological ordering. We then check if, for every node V_i in the topological ordering, there is an edge between V_i and V_{i+1}. I claim this will only be true if there is a Hamiltonian path. We need to visit all nodes $v_1 \dots v_n$ in an order. In a topological ordering, edges can never point backward. Thus if we take an edge (v_1, v_m) for $m < n$, we have no way of reaching the nodes in between v_1 and v_m. Because we cannot go backward, the only way we can form a Hamiltonian path on the equivalent topological ordering is to take the edge from one vertex to the ~~nearest~~ next adjacent vertex every time. If there are a nodes and b edges, we can construct a topological ordering in $O(a+b)$, and check whether each vertex has an edge to the next adjacent vertex in $O(b)$, so we have linear runtime of $O(a+b)$.

Name:

Student ID:

4. **[16 points]** Given strings $x = x_1x_2x_3...x_n$ and $y = y_1y_2y_3...y_m$ the *longest common substring (LCS)* is the identical substring (of sequential matching characters) in x and y of longest length. Show $O(mn)$ algorithm to find the length of LCS. Prove it is correct.

define a maximum substring of length 0

for every character x_i in x

or x or y terminates

(1) iterate through y until a character $y_i == x_i$ is found

(2) continue checking x_{i+k} with y_{i+k} until the characters do not match

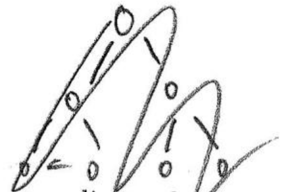
(3) if the length of this is longer than the maximum substring, the length of the new maximum is $k-1$, and update the substring

Repeat 1-3 from y_{i+k+1}, reset x_i to x_i

return the substring and length

For each n characters in x, we only consider the m characters in y once, so we have a runtime of $O(mn)$

This algorithm is correct. Every LCS must start with some ~~common~~ first common character. Thus we check ~~it every charact~~ ~~it on~~ every LCS that could start for every x_i in $x_1x_2...x_n$. Then we walk through x and y simultaneously until the substrings no longer match. Consider what would need to happen if ~~we~~ a longer substring indeed existed, ~~Then the last charact~~ but our algorithm was incorrect. By definition the last character ~~must match~~ of the substring must match. But our algorithm only terminates when the last charachter does not match, or one string terminates, in which there can be no longer substring. Thus our algorithm would have found this substring ∎



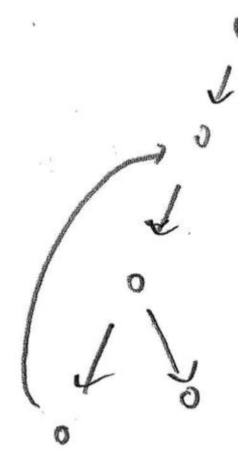
Name:

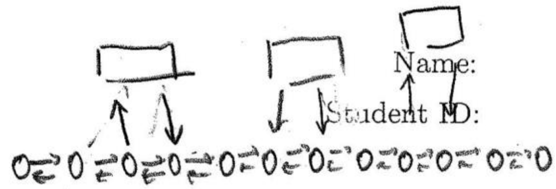
Student ID:

5. **[16 points]** Given a directed graph $G = (V, E)$ on n nodes and m edges describe $O(m + n)$ algorithm to find the length of shortest length cycle (if one exists).

We can find ~~an~~ shortest length cycle by using a simple adoptation of depth ~~breadth~~ first search. Recall that we construct a tree T based on G using DFS by choosing an arbitrary node, ~~then considering the neighbors of this node, then the neighbors of this neighbor, until the entire~~ then ~~considering~~ following a path starting at this node as far as we can go, ~~only~~ backing up when necessary. The difference is that when normally, if we reach an edge ~~we~~ incident on (u,v) we could have added but it would create a cycle, ~~instead, we count~~ instead of ignoring it, we count the distance of u to the least common ancestor of (u,v) and the distance of v to this least common ancestor, If this sum is shorter than the current shortest length cycle, this is the newest shortest length cycle.

We choose DFS because this is the natural analogue of following a path until it forms a cycle or terminates, and we want to consider cases where cycles are formed, ~~Perhaps a concern is that~~ Thus by using DFS, we consider all possible paths ~~from~~ starting at a node and ending because the path become a cycle at some point, or the path terminates, thus we considered all cycles. We can implement DFS in O(m+n) time, thus we have met the time constraint. Note that we also should check if some vertexes are not in the tree, as it is possible they are in a different component or unreachable from our first node. We repeat the process for these unincluded nodes and compare minimums to find the true minimum.





Name:

Student ID:

6. [**20 points**]

(a) [**15 points**] Recall that in class we showed a reduction from 3-SAT to Hamiltonian Cycle (in directed graphs). Explain how you would modify it to make a reduction from 20-SAT (e.g. every clause has at most 20 literals) to Hamiltonian Cycle problem. Does the number of nodes change? Does the number of edges change? If 20-SAT has n clauses and m variables, how many vertices and directed edges does your HC graph in the reduction has?

(b) [**5 points**] Someone shows that problem X is in NP and poly-time reducible to Graph 3-Coloring (i.e. $X \leq_p$ *3-COLORING*). Is it true that X must be NP-Complete? If yes, justify your answer, if no, show a counter-example.

a ~~Recall that our gadget was composed~~ of m rows of 3n+3 ~~nodes in the~~ case of 3-SAT. ~~This is because we have 3 node.~~

for each

Recall that our gadget was composed of m rows of 3n+3 nodes. We actually do not have to do much adaptation from 3-SAT. Like 3-SAT, for 20-SAT, we still have m rows of 3n+3 nodes. The only difference is, in 3-SAT each clause had 6 edges attached to its node, one directed edge in and one directed edge out for each literal in the clause. In the case of 20-SAT, each added clause node has 40 edges, and we connect these in the same way as we did in 3-SAT so that a traversal from left to right along the ~~clause line in~~ line m means an assignment of m to 1 and an assignment of 0 for right to left. The only difference is that each clause is now ~~on the pat~~ connected on ~~a path~~ 20 lines instead of 3.

We have $2 + m(3n+3) + n$ vertices. This is because we have a start and end node, plus $m(3n+3)$ vertices for each row, plus n additional vertices for clauses. We have $22m + 40n + 4m$ edges. This is because each clause has 40 attached edges, each line m has 22 edges, and each line uses 4 edges to connect to the next line. ~~The~~ Notice the number of nodes is unchanged, but edges are

b No. In fact, for X to be NP-complete, the reverse would have to be true, namely $X \leq_p$ 3-coloring. For example,